Voting Simulation Program System Guide

Original Code by Duncan Buell

System Guide and code modifications by:

Michael Cantwell

Robert Carff

Anthony Frazier

Carson Kaylor

Sebastian Martin

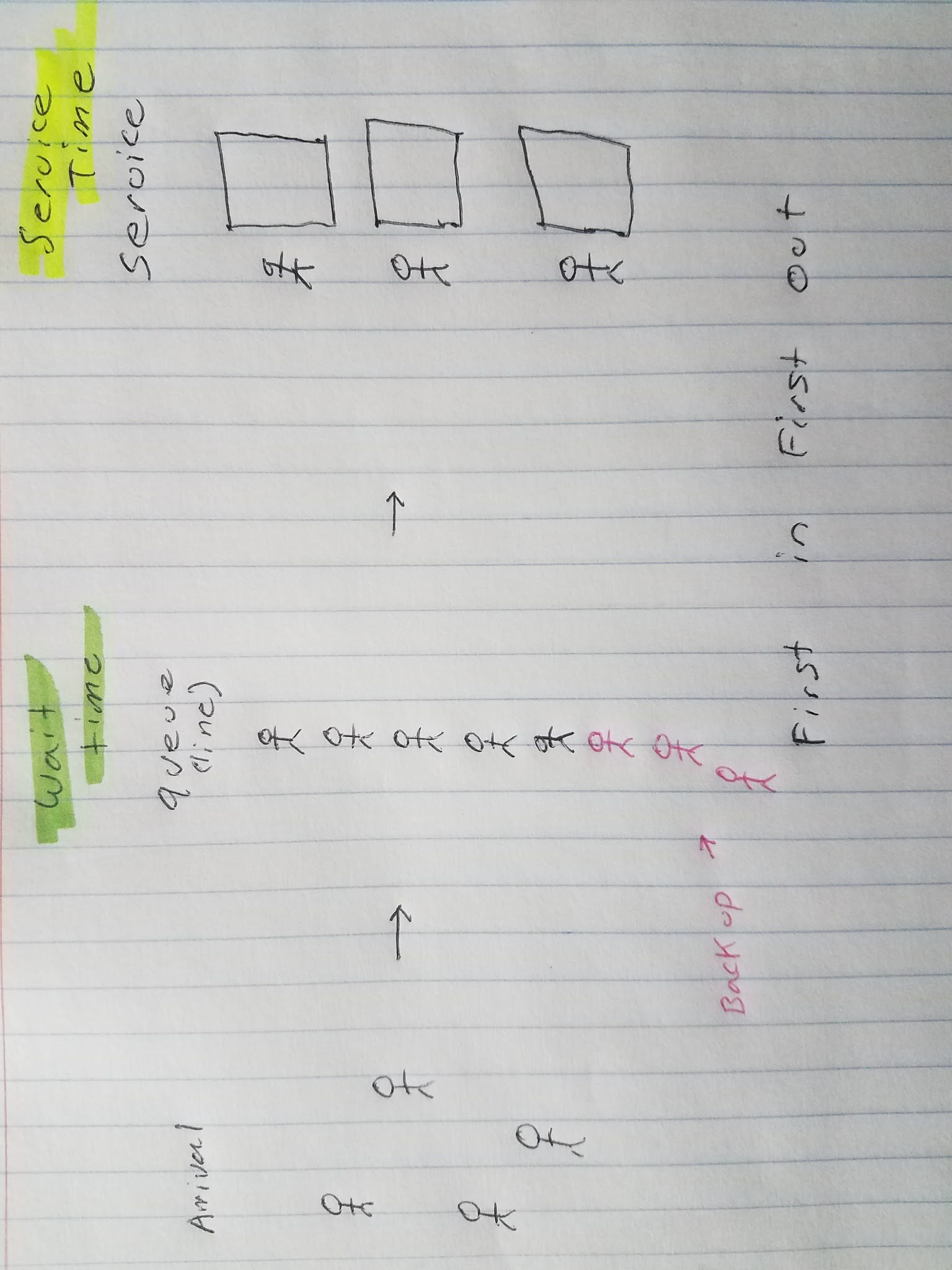
**-Overview-**

The two important ideas presented in this code are service time and wait time, how can we produce data explaining the connections between these two? How do they change based on different scenarios?

We assume this simulation has:

* **Input source** (Arrival of people)
  + Individual voters will arrive (no busses with massive increases in queue time)
  + Uses a Gaussian Distribution
* **Queue** (the time people wait)
  + Changes based on line length
  + Assuming people will continually arrive the entire day
  + This allows for analysis of fluctuation of arrival times (backups and empty lines)
  + We limit this by using variables such as **election\_day\_length\_hours\_**  ect.
* **Queue discipline** (assuming “infinite” queue)
  + People will be picked in a first-come-first-serve kind of fashion
* **Service Mechanism** (voting stations)
  + Time it takes people to vote
  + How people vote/Where people vote

In order to create each of the above sections, we use classes to break each entity apart, and then use a main method to run each entity together, forming a **main.cc** that creates a new instance of **Configuration: config** and **Simulation: simulation.**

The simulation will output multiple scenarios by changing the above explaining how the lack in service stations can exponentially increase queue times - thus letting the user know that more service stations are required.

**Example Images:**

**-Variables Used-**

**main.h**: none utilized.

**main.cc**: Contains dummy strings for the variables: **pct\_filename**, **log\_filename**, **out\_filename**, **outstring**, **timecall\_output**. It has declarations for **out\_stream**, and two Scanners: **config\_stream** and **pct\_stream**. It creates a new instance of **Configuration**: **config**, a new instance of **Simulation**: **simulation**, and a new instance of **MyRandom**: **random**. Throughout the course of **main.cc**, and in fact all .cc files, we utilize **Utils** and the variable **kTag** to keep track of where we are in the program, writing to our **logfilename**.

**configuration.h**: Includes dummy values for two constant int variables, **kDefaultSeed** and **KDummyConfigInt**. It also contains a dummy value for constant double variable **kDummyConfigDouble**. It assigns the value **kDefaultSeed** to the public variable **seed\_**.It also assigns the value **kDummyConfigInt** to the following public int variables: **election\_day\_length\_hours\_**, **election\_day\_length\_seconds\_**, **time\_to\_vote\_mean\_seconds\_**, **max\_expected\_to\_simulate\_**, **min\_expected\_to\_simulate\_**, **wait\_time\_minutes\_that\_is\_too\_long\_**, and **number\_of\_iterations\_**. It assigns **kDummyConfigDouble** to **arrival\_zero\_**, and finally establishes two vectors **actual\_service\_times** (int) and **arrival\_fractions** (double).

**configuration.cc**: Creates a string **line** to be used by the Scanner that is passed in, also opens up a ScanLine **scanline**. **seed\_**, which was declared in the header file,is filled in with inputted data along with **election\_day\_length\_hours\_**, **election\_day\_length\_seconds\_**, **time\_to\_vote\_mean\_seconds\_**, **max\_expected\_to\_simulate\_**, **min\_expected\_to\_simulate\_**, **wait\_time\_minutes\_that\_is\_too\_long\_**, and **number\_of\_iterations\_**. It fills in **arrival\_zero\_** with inputted data too. The vector **arrival\_fractions** is filled in as well. A new Scanner **service\_times\_file** is created and is used to fill in the other vector **actual\_service\_times\_**. And a string **s** is used in the ToString function.

**onepct.h**: Includes dummy variables **kDummyDouble**, **kDummyInt**, and **kDummyString**. It declares and assigns dummy values to **pct\_expected\_voters\_**, **pct\_expected\_per\_hour\_**, **pct\_minority\_**, **pct\_name\_**, **pct\_number\_**, **pct\_turnout\_**, **pct\_stations\_**, and **pct\_num\_voters\_**. It also declares **wait\_dev\_seconds\_** and **wait\_mean\_seconds\_**.A set of ints **stations\_to\_histo\_**, and a vector of ints **free\_stations\_**. Several multimaps of int keys and **OneVoter** data are made: **voters\_backup\_**, **voters\_done\_voting\_**, **voters\_pending\_**, and **voters\_voting\_**.

**onepct.cc**: In the **ComputeMeanAndDev** function it creates the int and double variables **sum\_of\_wait\_times\_seconds** and **sum\_of\_adjusted\_times\_seconds**. It initializes a multimap with int key and **OneVote**r data called **iter\_multimap**. It initializes **wait\_mean\_seconds\_** based on a calculation involving **sum\_of\_wait\_times\_seconds** and **pct\_expected\_voters**. It initializes **sum\_of\_adjusted\_times\_seconds**, and also **wait\_dev\_seconds**. In the **CreateVoters** function, we initialize int variables: **duration**, **arrival**, **sequence**, **percent**, and string **outstring**. We also initialize double **lambda**, int **interarrival**, and **durationsub**, all which utilize random to generate **RandomExponentialInt** or **RandomUniformInt**. In the **DoStatistics** function, we initiate a map, with key int of type int, **wait\_time\_minutes\_map**, and a multimap with key int of type **OneVoter iterator iter\_mulitimap**. We utilize int variables: **toolongcount**, **toolongcountplus10**, **toolongcountplus20**. In **RunSimulationPct**, we initialize **min\_station\_count** and **max\_station\_count**, both of type int. We also initialize a bool sentinel value **done\_with\_this\_count**, map of type int with int key **map\_for\_histo**, int **numbers\_too\_long**, int **time\_lower**, **time\_upper**, **voters\_per\_star**, **count**, and **count\_divided\_ceiling**. In **RunSimulationPct2**, we initialize int **second**, bool **done**, and multiple **auto iters**. We also utilize **which\_station** and **leave\_time** both of type int. In the **ToString** and **ToStringVoterMap**, we utilize string **s** to return a formatted string.

**onevoter.h**: Contains a dummy constant int variable **kDummyVoterInt**, and multiple private int variables including: **sequence\_**, **time\_arrival\_seconds\_**, **time\_done\_voting\_seconds\_**, **time\_start\_voting\_seconds\_**, **time\_waiting\_seconds\_**, and **which\_station\_**.

**onevoter.cc**: The parameterized constructor for OneVoter will assign values to **sequence\_**, **time\_arrival\_seconds\_**, **time\_start\_voting\_seconds\_** = **0**, **time\_vote\_duration\_seconds\_** and **which\_station\_** = **-1** (magic number) in that order. The **AssignStation** function will assign a station number and start time to a voter, utilizing the int variables **station\_number** and **start\_time\_seconds**. The **ConvertTime** function’s use of int **hours**, **minutes**, **seconds**, and string **s** produces a formatted string with the standard hour:minute:second format of time. Finally, in the **ToString** and **ToStringHeader** functions, string **s** is utilized to produce a formatted string of information.

**simulation.h**: The only variable declaration in **simulation.h** is a map, with an int key and data type **OnePct** called **pcts\_**.

**simulation.cc**: Inside the function **RunSimulation**, we are utilizing an incrementing int variable **pct\_count\_this\_batch\_** that counts the number of precints parsed in this batch of data. We utilize an iterator **iterPct** to iterate over all of the precincts, and assign the int variable **expected\_voters** to the variable we obtain from the accessor **GetExpectedVoters**, contained in the **OnePct** class. Finally, the standard **ToString** function utilizes string **s** to return a formatted string of text.

**-Step by Step Process-**

Executing the program will require execution of the file along with 4 filename arguments: **configfilename**, **pctfilename**, **outfilename**, **logfilename**. (the name of the configuration file, the name of the precinct file, the name out of the output file, and the name of the log file). Execution will then start in **main.cc**.

Execution begins by reading the file names from the execution arguments: **configfilename**, **pctfilename**, **outfilename**, **logfilename**; and assigning to the above variables. We then open the file **config\_filename** into the **config\_stream** and parse the data with **Configuration.ReadConfiguration**.

In the **Configuration.ReadConfiguration** function, we parse the data as one whole line, that we then tokenize over to obtain the following, in order: **seed\_**, **election\_day\_length\_hours\_**, **time\_to\_vote\_mean\_seconds\_**, **min\_expected\_to\_simulate\_**, **max\_expected\_to\_simulate\_**, **wait\_time\_minute\_that\_is\_too\_long\_**, and **number\_of\_iterations\_**. During the parsing of this data, we determine **election\_day\_length\_seconds\_** by calculating the seconds based on the hours obtained through parsing the data. On the following line, we expect to find **arrival\_zero\_** as a double. We then parse doubles until the end of the line, pushing all values found into the vector **arrival\_fractions\_**. We next open the hardcoded text file **“dataallsorted.txt”**, and push all integers found within this file into the vector **actual\_service\_times\_**.

After the configuration has been read, we close the **config\_stream**. We utilize **Configuration.ToString** to convert the config file into formatted text, which is output via **out\_stream** to our **out\_filename**.

In the **Configuration.ToString** function, we return a formatted string of the variables in configuration in the following format:

CONFIG: RN seed: **seed\_**

CONFIG: Election Day length: **election\_day\_length\_seconds\_** = **election\_day\_length\_hours\_**

CONFIG: Time to vote mean: **time\_to\_vote\_mean\_seconds\_** = (**time\_to\_vote\_mean\_seconds\_** / 60) minutes

CONFIG: Min and max expected voters for this simulation: **min\_expected\_to\_simulate\_ max\_expected\_to\_simulate\_**

CONFIG: Wait time(minutes) that is ‘too long’: **wait\_time\_minutes\_that\_is\_too\_long\_**

CONFIG: Number of iterations to perform: **number\_of\_iterations\_**

CONFIG: Max service time subscript: **GetMaxServiceSubscript**

CONFIG: **arrival\_zero\_**

Finally, we parse over the **arrival\_fractions\_** vector and retrieve the information, and format that below the above table. We then generate a new random by calling **MyRandom** where we utilize **seed\_** from the configuration file.

In MyRandom, we utilize seed\_ to create a new instance of MyRandom by utilizing the parameterized constructor.

Once we return to main.cc, we then read the precinct data into the **pct\_stream** by opening the **pct\_filename**, and utilizing **Simulation.ReadPrecincts**.

In the **Simulation.ReadPrecincts** function, we tokenize over the **pct\_filename**(**infile**), and create a new precinct **OnePct new\_pct**. We then utilize the **OnePct.ReadData** function, and after parsing this data, we push this new precinct into the **pcts\_** map.

In the **OnePct.ReadData function**, we parse over the **infile**, and obtain the following values: **pct\_number\_**, **pct\_name\_**, **pct\_turnout\_**, **pct\_num\_voters\_**, **pct\_expected\_voters\_**, **pct\_expected\_per\_hour\_**, **pct\_stations\_**, **pct\_minority\_** as well as **stat1**, **stat2**, and **stat3**. We push these three **stat**(s) into the **stations\_to\_histo\_** set.

We then close the **pct\_stream**. Now that we have set up all of the data, we now utilize our instance of **Simulation**, **simulation**, to actually simulate our data with **Simulation.RunSimulation**. We utilize the **config** and **random** data we have read, and passing along our **out\_stream** for output to perform this simulation.

In **Simulation.RunSimulation**, the real work begins. We initialize a counting variable **pct\_count\_this\_batch** to count how many precincts we have simulated, in this batch. We then have our overall for loop that’s purpose is to use the iterator **iterPct** to iterate over the entire **pcts\_** map. We create a temporary precinct, **OnePct pct**, and copy the precinct returned in this case of **pcts\_**. We obtain local variable **expected\_voters** utilizing the **OnePct.GetExpectedVoters** function, which works exactly as expected. Now we hit our first major conditional: if the **expected\_voters** is less than or equal to the **min\_expected\_to\_simulate** that we obtain from **config**, or if **expected\_voters** is greater than the **max\_expected\_to\_simulate,** also from **config**, we continue into the loop, else we allow our iterator to advance one precinct.

If we continue in the above conditional, we utilize the **OnePct.ToString** function to return the formatted string of precinct information as follows:

SIM: **pct\_number\_** **pct\_name\_** **pct\_turnout\_** **pct\_expected\_voters\_** **pct\_expected\_per\_hour\_** **pct\_stations\_** and **pct\_minority\_** HH

We then iterate over the set **stations\_to\_histo\_** and return this information as a formatted string at the end of the above line, followed by HH.

We increment our counter **pct\_count\_this\_batch**, and call the **OnePct.RunSimulationPct** on our pct instance, passing along **config**, **random**, and our **out\_stream**.

In the **OnePct.RunSimulationPct**, we calculate the variable **min\_\_station\_count** based on **pct\_expected\_voters** and **config.time\_to\_vote\_mean\_seconds** and the **config\_.election\_day\_length\_hours\_.** We utilize a sentinel check to ensure that our **min\_station\_count** is at least one (we’re not going to get very far with zero voting stations). We also calculate our **max\_station\_count** based on our **min\_station\_count** and our **config.election\_day\_length\_hours**. We declare a sentinel bool, **done\_with\_this\_count** and set it to false. We set up a for loop, using a counting variable **stations\_count** and set it initially equal to **min\_station\_count**, and the exit condition is when **stations\_count** is equal to **max\_station\_count**. If we reach this top-most portion of the loop with the **done\_with\_this\_count** equal to true, we break and immediately exit the loop. After this break exit condition, we set **done\_with\_this\_count** equal to true, and we create a new map of integers, using an int key, called **map\_for\_histo\_**. We utilize a standard **ToString** to cover our **station\_count** (int) to a string to be easily passed to our outstream, and we continue.

After outputting the above formatted text, we start another, nested, for loop. This loop starts at **iteration** is equal to zero, and our exiting condition is **iteration** equals **config.number\_of\_iterations\_**. We then call the **OnePct.CreateVoters** function, passing along **config**, **random**, and our **out\_stream**.

In the **OnePct.CreateVoters** function, we initialize the following variables: **duration**, **arrival**, and **sequence** of type int, a double **percent**, and a string **outstring**. We clear our **voters\_backup\_**, and initialize **sequence** equal to zero, just in case. We set **percent** equal to **arrival\_zero\_** from **config**, and initialize **voters\_at\_zero** as an int calculated by **percent** and **pct\_expected\_voters\_**. We then initialize a **voter** with a for loop, with exiting condition when **voters** equal **voters\_at\_zero**. Inside this loop, we set **duration** equal to **config.actual\_service\_times** at **durationsub**. We create a new voter **OneVoter one\_voter** passing **sequence**, **arrival**, and **duration**.

In the **OneVoter** constructor we assign the passed values, **sequence**, **arrival**, and **duration** to local variables **sequence\_**, **time\_arrival\_seconds\_**, and **time\_vote\_duration\_seconds\_**. We initialize two new local variables, **time\_start\_voting\_seconds\_** and **which\_station\_**.

Now that we have initialized a new voter, we return to **OnePct.CreateVoters**, and we insert this voter into the **voters\_backup\_** map, increment **sequence**, and increment our for loop. Once we exit the voter creation loop, we then put the voters in line. We do this by utilizing a for loop, starting with **hour** equal to zero, and our exit condition is when **hour** equals **config.election\_day\_length\_hours\_**. Inside this loop, we set **percent** equal to **config.arrival\_fractions** at **hour**, and we declare a new variable, **voters\_this\_hour** which is calculated by **percent** and **pct\_expected\_voters\_**. We utilize a calculation to determine if we need to add more voters this hour. If so, increment **voters\_this\_hour**, and if not, then skip to the next step inside the loop. We initialize a new variable, **arrival**, and calculate this based on **hour**. We then start a nested for loop, with our exit condition being that new variable **voter** is equal to **voters\_this\_hour**. Inside this new loop, we initialize a new double variable **lambda**, which is calculated based on **voters\_this\_hour**. We initialize a new int variable **interarrival**, which we utilize **MyRandom.RandomExponentialInt** to produce, sending it our variable **lambda**.

In the **MyRandom.RandomExponentialInt** function, we assert that the **lambda** sent to us is greater than or equal to zero, and if so we continue, else we break. We generate an int **r** based on an **exponential\_distribution** based on **lambda**, and we round this to the nearest whole value (int) and return this value.

We calculate **arrival** based on **arrival** and **interarrival**, and declare another new variable **durationsub**, which is calculated based on **MyRandom.RandomUniformInt** and **config.GetMaxServiceSubscript**.

In the **MyRandom.RandomUniformInt**, we utilize the **lower** bound hardcoded value zero, and **config.GetMaxServiceSubscript** as our **upper** bound. We assert that **lower** is less than or equal to **upper**, and only if this is true do we proceed. We generate a **uniform\_int\_distribution** **distribution** based on our **lower** and **upper** bounds, and use this to generate an int, **r**, which is returned.

Once we return from creating a new random, we calculate **duration** based on **config.actual\_service\_times** at **durationsub**, we insert this voter into the **voters\_backup\_** and increment sequence.

Once we return from the **CreateVoters** function, we set **voters\_pending\_** to our **voters\_backup\_**, and reset our **voters\_voting\_** and **voters\_done\_voting**. We then call **OnePct.RunSimulationPct2**, sending along **stations\_count**.

In the **RunSimulationPct2** function we reset all **free\_stations\_**, and start a new for loop with our exit condition that new variable int **i** is equal to **stations\_count**. We utilize **i** to generate free voting booths that we push into the **free\_stations\_** vector. After setting up our free stations, we reset our **voters\_voting\_** and **voters\_done\_voting\_**. We initialize two new variables, int **second** and bool **done**, and use **done** to start a new sentinel controlled while loop. While **done** is equal to false (**!done**), continue to run the loop. Inside the loop is a for loop, controlled by an iterator **iter**, that will iterate over the **voters\_voting\_**, with the exit condition being when **iter** has reached the end of **voters\_voting\_**. We establish a new instance of **OneVoter**, **one\_voter**, and assign it to the **second** thing that **iter** is pointing at. We initialize a new variable **which\_station**, which uses an accessor **one\_voter.GetStationNumber** to obtain the station number of the available station. We push the station number back into **free\_stations\_**, and insert the voter into the **voters\_done\_voting\_ map**. We then erase the **voters\_voting\_**, and create a new vector of maps of **OneVoter** named **voters\_pending\_to\_erase\_by\_iterator**.

Still in **RunSimulationPct2**, we start a new for loop, still contained within our sentinel **!done** check. This loop is an iterator named **iter** whose ending condition is when iter reaches the end of **voters\_pending\_**. There is an if condition which ensures that the **next\_voter** time of arrival isn’t less than the next position of the iterator, and as long as this is true, we assign the voter to the voting station that is available at position zero in **free\_stations\_** and assign this station to the new variable **which\_station**. We remove this station from the **free\_stations\_**, and assign the voter to this free station via **next\_voter.AssignStation**.

In **OneVoter.AssignStation** we initialize the voter to a **station\_number** by declaring the local variable **which\_station\_** and also parse **time\_start\_voting\_seconds**, **time\_done\_voting\_seconds\_** and **time\_waiting\_seconds** by calculations.

Once we return from **OneVoter.AssignStation**, we initialize a new variable **leave\_time** of type int, and assign it to the value accessed by **next\_voter.GetTimeDoneVoting**. We add to our **voters\_voting\_ map** the **leave\_time** and **next\_voter**, and then add the voter that has completed voting to **voters\_pending\_to\_erase\_by\_iterator\_**.

Once we return from the **RunSimulationPct2** function, we assign int **number\_too\_long** to the number returned by calling **DoStatistics**, passing along **iteration**, **config**, **stations\_count**, **map\_for\_histo\_** and **out\_stream**.

In the **DoStatistics** function we initialize a map **wait\_time\_minutes\_map** and a multimap **iter\_multimap**. We then use a for loop to get the **wait\_time\_minutes** from each voter and insert that into the respective maps. Then a second for loop checks if the **waittime**’s are above an amount determined earlier by **config.wait\_time\_minutes\_that\_is\_too\_long\_** and if so increases the **waitcount**. We then call the **ComputeMeanAndDev()** function.

In the **ComputeMeanAndDev** function we start by creating **sum\_of\_wait\_times\_seconds** and **sum\_of\_adjusted\_times\_seconds**. We initialize the multimap **iter\_multimap**. We then iterate through the map in a for loop and initialize instances of **OneVoter**, **voter,** then take and add **voter.GetTimeWaiting()**, which returns the time waiting variable stored in that instance of voter, to the running total, **sum\_of\_wait\_times\_seconds**. We then compute **wait\_mean\_seconds\_**. We then use another for loop to iterate through the map and change the waiting time to an adjusted value and then collect a running total again called **vsum\_of\_adjusted\_times\_seconds**. Then last **wait\_dev\_seconds\_** is calculated.

Once we return to the **DoStatistics** function we begin to format all of the statistics for the output. Using **Utils::Format()** we start to form the output string, **outstring**, with the **kTag**, **iteration**, **pc\_number\_**, **pct\_name\_**, **pct\_expected\_voters\_**, **station\_count**, **wait\_mean\_seconds\_**, **wait\_dev\_seconds\_**, and the **tolongcount** data.We then use **Utils::Output()** to output **outstring** by passing in **outstring**, **out\_stream**, and **Utils::log\_stream**.Last we clear **wait\_time\_minutes\_map** and then return **toolongcount**.

Once we return from the **DoStatistics** function, if **number\_too\_long** is greater than zero, we reset **done\_with\_this\_count** to false, and jump to the next **iteration** in our inner loop. Once we exit our **iteration** inner loop, we clear **voters\_voting\_** and **voters\_done\_voting\_**, and output “toolong space filler”. We then jump into an if conditional, where we check to see if the **stations\_to\_histo\_.count(stations\_count)** is greater than zero, if so, we output HISTO and convert **stations\_to\_histo\_** to a normal string via normal ToString method for a set.

In **this->ToString** we first create the output string **s**. We then add on data using **Utils::Format()**. First we add **pct\_number\_**, then **pct\_name\_**, **pct\_turnout\_**, **pct\_num\_voters\_**, **pct\_expected\_voters\_**, **pct\_expected\_per\_hour\_**, **pct\_stations\_**, and **pct\_minority\_**. Then “HH” is added. We then use a for loop to iterate through the map **stations\_to\_histo\_** and use **Utils::Format()**  to add each instance to the output string **s**. We then add “HH” again and return **s**.

Once we return from the **this->ToString** function, we output

HISTO STATIONS: stations\_count

We then initialize int variables **time\_lower** and **time\_upper**, which mark the start and end of the **map\_for\_histo** locations, respectively. We initialize **voters\_per\_star** is equal to one, and run into an if statement: if **map\_for\_histo** at **time\_lower** is greater than fifty, then we recalculate **voters\_per\_star** based on **map\_for\_histo** at **time\_lower** and **config.number\_of\_iterations**. We ensure that **voters\_per\_star** has not fallen below or equal to zero, and if it has, we reset it equal to one. We then jump into another for loop, starting at int **time** is equal to **time\_lower**, with our exit condition being when **time** is equal to **time\_upper**. We access the **map\_for\_histo** at **time**, and set new int variable **count** equal to that value. We then calculate the double variable **count\_double** based on **count** and **config.number\_of\_iterations\_**. We also calculate **count\_divided\_ceiling** based on **count\_double** and **voters\_per\_star**. We determine the new variable **stars**, of type string, as **count\_divided\_ceiling**, and **\***. We finally output HISTO and end our if and outer for loops.

Now that we are finally back to **Simulation.RunSimulation**, we output:

PRECINCT COUNT THIS BATCH: pct\_count\_this\_batch

Finally, we return to our main.cc and we close our **out\_stream** upon ending execution.